Faculty of Engineering of the University of Porto

Project Management System

Databases - Bachelor in Informatics and Computing Engineering

Group 401 - Students & Authors:

Daniel Pinto up202304957@up.pt

Felipe Ramos up202303505@up.pt

Jakub Kaminski up202408623@up.pt

Index

4.1 First Submission: Conceptual Modelling

4.1.1 Domain Definition

A PMS (Project Management System) serves as the “spinal-cord” of every company. It helps in managing multiple projects, departments, employees, business partners, etc. In that way, it allows the company to operate efficiently, controlling every area, generating reports, saving contacts and managing all the projects in an organized manner. A good PMS always tries to replicate the stages of the clients operations, be it with different project steps, different interconnected sales and acquisition of material, how the clients relate to the company employees and how those employees are part of different departments.

The most important entity in a PMS is the Company that produces and sells the products, it being the software owner that will provide all the input to the PMS. The Company, as the center point of the software (and the operation) is indirectly connected to all entities present on the software, as it is also, in the real world, connected to all operations it executes. The system enables the company to organize all its workforce on Departments and areas, creating a solid organizational structure that will later be important not just for the daily management of the company but also for strategic decisions such as the allocation of resources and personnel for each Project or even the definition of internal workflows that will define how the company operates on “extra-software” level.

Each department in a company has their own tasks and objectives. A PMS is the key to manage all departments and its employees and mainly, differentiate the sales from those many departments for later analysis, always keeping track of the budget given to each department and their adherence to their proposed strategies. Keep in mind that some Departments might not be related directly to sales, but still, it’s important to keep track of their performance through many different metrics (later defined using the data available on the PMS).

A business is nothing without Sales, therefore a PMS also does not function without them. Sales are the cornerstone of every company and a PMS will help, as it helps with each department, to organize every sale, so the company doesn’t lose money due to inefficiency and lack of internal planning. With each sale, a PMS will store the related documents, assign each sale to an Employee/Department and keep track of the profits (or losses) from each sale, creating a whole history of this sale and how it went, creating a comprehensible timeline that can be used for later analysis.

The employees are also an important entity in a PMS as without them the whole operation would freeze. A PMS is important to ensure how each employee is involved in the operation, in which departments they’re working and tracking the success of projects they’re involved in.
Ultimately, as the name suggests a Project Management System is helpful in turning the management of companies on a seamless task, by connecting the company, its

departments, employees, sales, financial documents, customers and general business partners on a robust information web.

4.1.2 Initial Conceptual Modeling

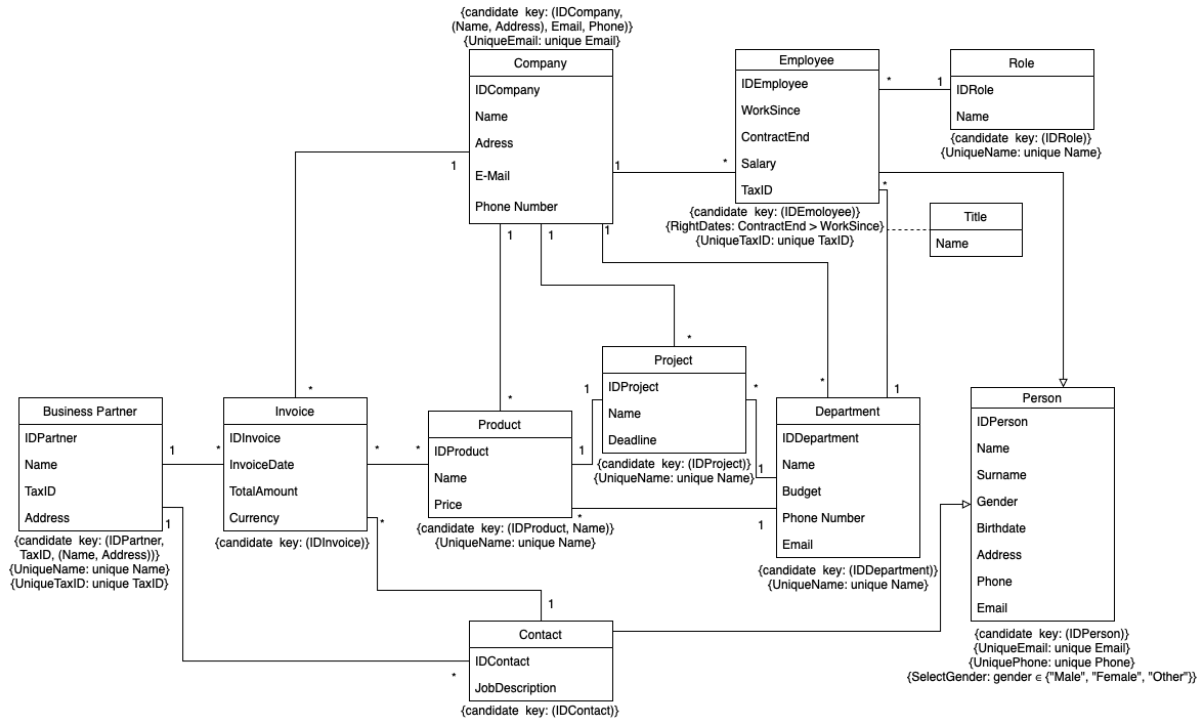

Image 1: Initial Conceptual Model

4.1.3 Generative AI Integration

This interaction was made with ChatGPT 4.o

The idea behind the initial prompt was to generate something completely different from our original approach, so we could decide what to introduce in our Model later on, turning the final conceptual model a mix between our work and the AI suggestion.

On our questions, we focused on understanding the reasoning behind the suggestions made, understanding why or why not to utilize the given information in our final version.

We finished by asking it to generate a comprehensible guide to recreate the diagram proposed.

Link to the whole interaction:
https://chatgpt.com/share/6749c606-6c80-8007-a29b-686fca9d016f

4.1.4 Final Conceptual Modeling

Our final conclusion, after the use of the AI model was to reimplement the class Sale and to change Project attributes, adding a StartDate and EndDate, while removing Deadline.

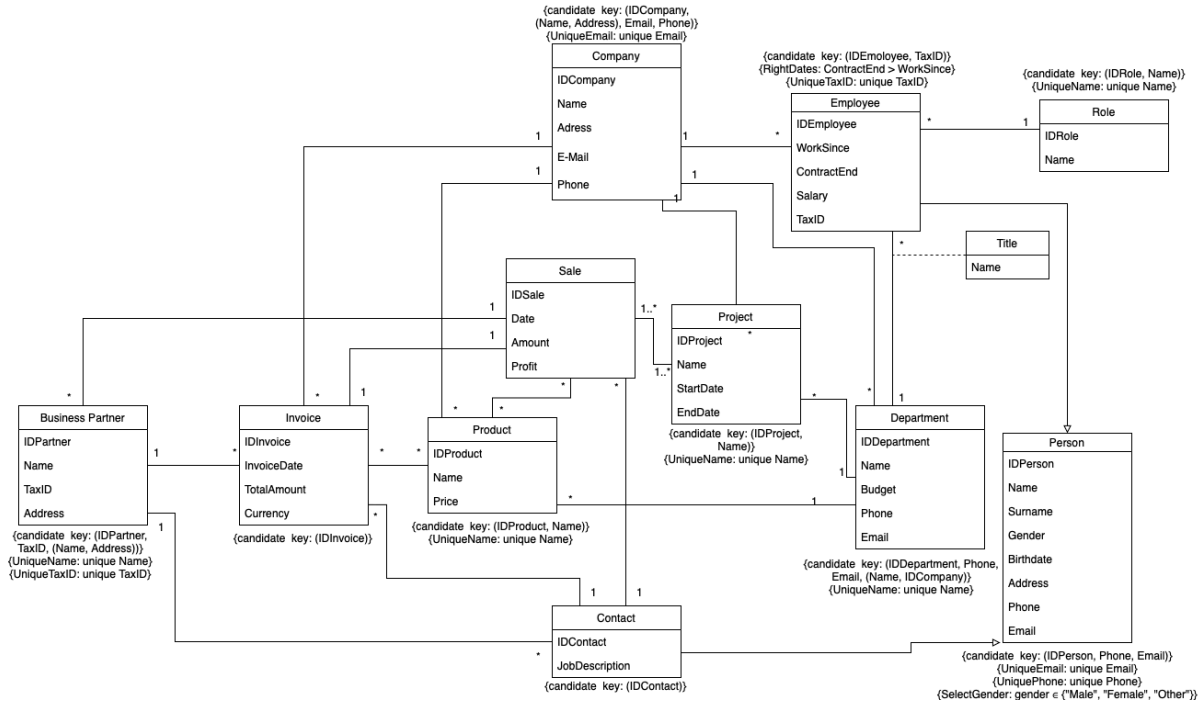

Image 2: Final Conceptual Model

5.1 Second Submission: Relational Modelling, Database Creation and Data Population

5.1.1 Refinement of the Conceptual Model

We changed the relation between Project and Sale, so it's one to many. It didn't make sense to only have Projects that continued a Sale, as a new Project might not have a Sale related to it. Also, it doesn't make sense to have a single Sale relating to multiple Processes.

Added Currency, so we could represent in which currency any sales were made.

We deleted the column Profit in the Sale table, as a Profit can only be calculated if you have the expenditures information. Therefore, we chose to keep the profit as something you can calculate by taking the information in a new table called Purchase.

Added Purchase so it is possible to calculate Profit and represent items and services purchased by the company. This Purchase table makes it possible to calculate the Profit for all Companies registered in our system.

We cut the relation between Invoice and BusinessPartner once they were already connected by the Sale, making it redundant.

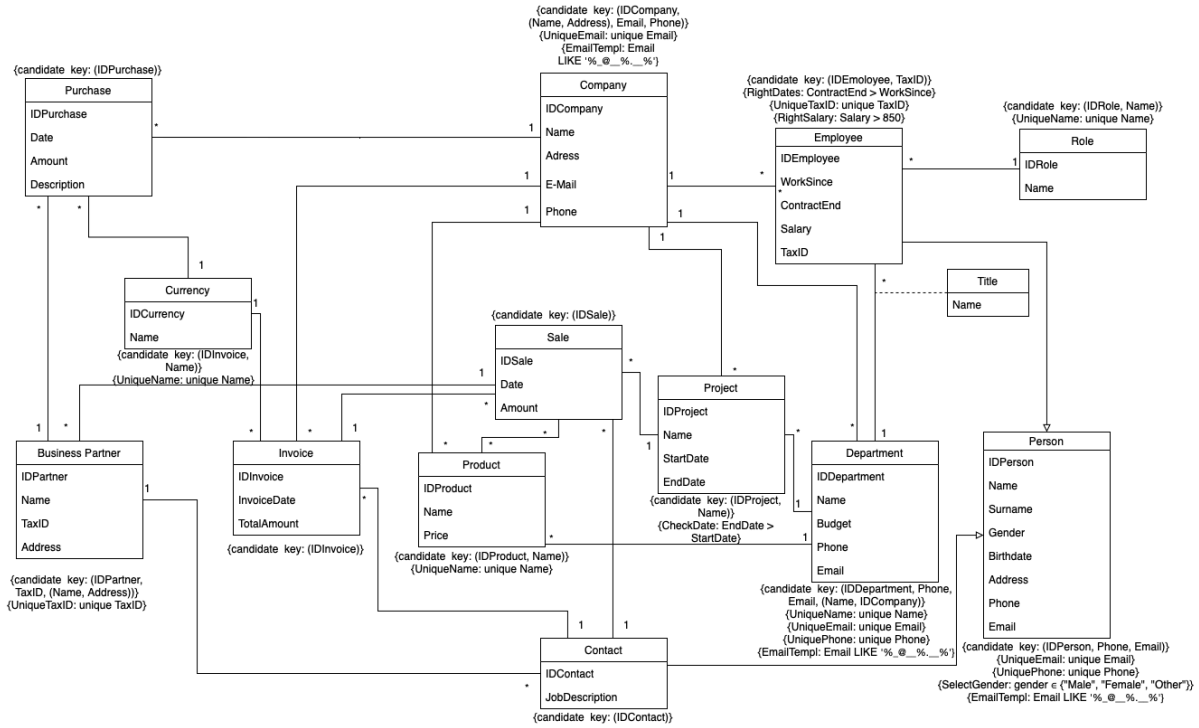

Image 3: Refined Conceptual Model

5.1.2 Initial Relational Schema

Since we still haven't made the decision to add the Purchase table here, our Initial Schema, without AI still does not include the Purchase table.

- BusinessPartner(IDPartner, Name, TaxID, Address, IDSale->Sale)
- Invoice(IDInvoice, InvoiceData, TotalAmount, Currency, IDCompany->Company, IDContact->Contact, IDBusinessPartner -> BusinessPartner)
- Product(IDProduct, Name, Price, IDDepartment->Department, IDCompany->Company)
- ProductSale(IDProduct->Product, IDSale->Sale)
- InvoiceSale(IDInvoice -> Invoice, IDSale -> Sale)
- Sale(IDSale, Date, Amount, Profit, IDContact->Contact, IDProject -> Project)
- Company(IDCompany, Name, Address, E-Mail, Phone)
- Contact(IDContact -> Person, JobDescription, IDPartner->BusinessPartner)
- InvoiceProduct(IDInvoice->Invoice, IDProduct->Product)
- Person(IDPerson, Name, Surname, Gender, Birthdate, Address, Phone, Email)
- Employee(IDEmployee -> Person, IDRole -> Role, WorkSince, ContractEnd, Salary, TaxID, IDCompany -> Company)
- Role(IDRole, Name)
- DepartmentTitle(IDEmployee -> Employee, IDDepartment -> Department, Name)
- Department(IDDepartment, Name, Budget, Phone, Email, IDCompany -> Company)
- Project(IDProject, Name, StartDate, EndDate, IDDepartment -> Department, IDCompany -> Company)

5.1.3 Generative AI Integration

To keep the report concise, only the changes on the last schema will be shown bellow:

1. On the Invoice table, IDBusinessPartner was deleted, as you can reach it through Contact, we also deleted Currency, which became its own table

Invoice(IDInvoice, InvoiceDate, TotalAmount, IDCurrency -> Currency, IDCompany -> Company, IDContact -> Contact, IDCurrency -> Currency)
Currency(IDCurrency, Name)

2. On the Sale table, we deleted Profit, as it is supposed to be made by a calculation between the amount spent and earned, which does not compete with this table.
Sale(IDSale, Date, Amount, IDContact -> Contact, IDProject -> Project)
3. Added purchase so it is possible to calculate profit
Purchase(IDPurchase, Date, Amount, Description, IDCompany -> Company, IDPartner -> BusinessPartner, IDCurrency -> Currency,)
4. DepartmentTitle was renamed to EmployeeDepartment
EmployeeDepartment(IDEmployee -> Employee, IDDepartment -> Department, Name)
5. Deleted IDSale from BusinessPartner. It's possible to find the Business Partner related to a sale through the Sale's contact
BusinessPartner(IDPartner, Name, TaxID, Address)

The following link contains the prompts and results that brought the changes above:
https://chatgpt.com/share/6745e5a6-a874-8007-81f1-f3f34452bdc2

5.1.4 Final Relational Schema

- BusinessPartner(IDPartner, Name, TaxID, Address)

- Invoice(IDInvoice, InvoiceData, TotalAmount, Currency, IDCompany->Company, IDContact->Contact, IDBusinessPartner -> BusinessPartner)

- Product(IDProduct, Name, Price, IDDepartment->Department, IDCompany->Company)

- ProductSale(IDProduct->Product, IDSale->Sale)

- InvoiceSale(IDInvoice -> Invoice, IDSale -> Sale)

- Sale(IDSale, Date, Amount, Profit, IDContact->Contact, IDProject -> Project)

- Company(IDCompany, Name, Address, E-Mail, Phone)

- Contact(IDContact -> Person, JobDescription, IDPartner->BusinessPartner)

- InvoiceProduct(IDInvoice->Invoice, IDProduct->Product)

- Person(IDPerson, Name, Surname, Gender, Birthdate, Address, Phone, Email)

- Employee(IDEmployee -> Person, IDRole -> Role, WorkSince, ContractEnd, Salary, TaxID, IDCompany -> Company)

- Role(IDRole, Name)

- DepartmentTitle(IDEmployee -> Employee, IDDepartment -> Department, Name)

- Department(IDDepartment, Name, Budget, Phone, Email, IDCompany -> Company)

- Project(IDProject, Name, StartDate, EndDate, IDDepartment -> Department, IDCompany -> Company)

5.1.5 Initial Analysis of Functional Dependencies and Normal Forms

For all tables we did the analysis step by step, from 1st Normal Form to Boyce-Codd Normal Form, identifying possible issues and correcting them on the go.

As an example, and to not repeat the same steps multiple times in the report, the BusinessPartner table will serve as a showcase of the analysis made to all tables.

Tables where changes were necessary will also be shown below.

BusinessPartner(IDPartner, Name, TaxID, Address)
- 1NF verification: has atomic values, impossible to have different data types in the same column, so it is 1NF
- 2NF: is in 1NF, candidate keys are: {IDPartner}, {TaxID}, {Name, Address}. Within them, there is no partial dependency.
- 3NF: is in 2NF, no non-prime, therefore there are no transitive dependencies
- BCNF: is in 3NF, all functional dependencies have a candidate key as determinant, therefore, it is in BCNF
-

Product(IDProduct, Name, Price, IDDepartment->Department, IDCompany->Company)
- 1NF: has atomic values, impossible to have different data types in the same column, so it is 1NF
- 2NF: is in 1NF, candidate keys are: {IDProject}, {Name}. There are no partial dependencies.
- 3NF: is in 2NF, but IDCompany is dependent on IDDepartment, meaning it isn't in 3NF

Project(IDProject, Name, StartDate, EndDate, IDDepartment -> Department, IDCompany -> Company)
- 1NF: has atomic values, impossible to have different data types in the same column, so it is 1NF
- 2NF: is in 1NF, candidate keys are: {IDProject}, {Name}. There are no partial dependencies.

- 3NF: is in 2NF, but IDCompany is dependent on IDDepartment, meaning it isn't in 3NF

The final conclusion, after identifying the dependencies was:

1. Take out Company, breaking the transitive dependency:
    - Product(IDProduct, Name, Price, IDDepartment -> Department)
    - Project(IDProject, Name, StartDate, EndDate, IDDepartment -> Department)

5.1.6 Generative AI Integration

We asked for a step by step guide on how to verify if a table is indeed in 3rd Normal form and Boyce-Codd Normal Form. After using the answer as a guide to review our original Relational Schema, we then asked it to make the review itself.

The AI review provided us with a useful descriptive analysis that made us believe that the schema developed on 5.1.5 was indeed correct, in 3rd Normal Form and also Boyce-Codd Normal Form.

The whole interaction can be checked on the link below:
https://chatgpt.com/share/6746345e-0254-8007-ae3b-6ebbb70a46ec

5.1.7 Final Analysis of Functional Dependencies and Normal Forms

After comparing our resulting schema with the AI's answers, we came to the conclusion that the first proposed analysis was indeed correct, and will serve as our final relational schema

Final schema:

- BusinessPartner(IDPartner, Name, TaxID, Address)
- Invoice(IDInvoice, InvoiceDate, TotalAmount, IDCurrency -> Currency, IDCompany -> Company, IDContact -> Contact)
- Product(IDProduct, Name, Price, IDDepartment -> Department)
- ProductSale(IDProduct -> Product, IDSale -> Sale)
- InvoiceSale(IDInvoice -> Invoice, IDSale -> Sale)
- Sale(IDSale, Date, Amount, IDContact -> Contact, IDProject -> Project)

- Company(IDCompany, Name, Address, Email, Phone)
- Person(IDPerson, Name, Surname, Gender, Birthdate, Address, Phone, Email)
- Contact(IDContact -> Person, JobDescription, IDPartner -> BusinessPartner)
- Employee(IDEmployee -> Person, IDRole -> Role, WorkSince, ContractEnd, Salary, TaxID, IDCompany -> Company)
- Role(IDRole, Name)
- EmployeeDepartment(IDEmployee -> Employee, IDDepartment -> Department, DepTitle)
- Department(IDDepartment, Name, Budget, Phone, Email, IDCompany -> Company)
- Project(IDProject, Name, StartDate, EndDate, IDDepartment -> Department)
- Currency(IDCurrency, CurrencyName)
- Purchase(IDPurchase, Date, Amount, Description, IDCompany -> Company, IDPartner -> BusinessPartner, IDCurrency -> Currency,)

5.1.8 Initial SQLite database creation

On the initial SQLite database creation, we started with the creation of tables and defined the constraints afterwards.

The constraints shown are related to the sizes of fields (TaxID for example), auto incrementation of Primary Keys, non null values on important fields (Name, Address, etc), definition of Gender, logical constraints on dates (starting dates shouldn't be after ending dates) and rules on minimum wages (Salary).

Constraints:
- Auto Increment PK
- Important Information not null
- Referencing of foreign keys
- TaxID has max number of chars to 14
- Person: Gender must be (Male, Female, Other)
- Employee (ContractEnd > WorkSince) (Salary>850)

5.1.9 Generative AI Integration

We used chatGPT 4.o to help define constraints and how to use them, from it we got different ways to verify emails and phone numbers.
By using Generative AI we expanded our constraints to also check if fields like email and phone number are filled correctly.

It was found that it is possible to achieve the same objective with different SQLite tools.

1. Using LIKE:
    - email TEXT NOT NULL UNIQUE CHECK (email LIKE '%_@_%._%'),

2. Using GLOB (normal Regex)
    - email TEXT NOT NULL UNIQUE CHECK (email GLOB '*@*.*' AND email NOT GLOB '*@*.' AND email NOT GLOB '*@*.*.' AND email NOT GLOB '*..*'),

    - phone_number TEXT NOT NULL CHECK (phone_number LIKE '+%[0-9]%' AND length(phone_number) >= 10), tax_id TEXT NOT NULL UNIQUE CHECK (tax_id GLOB '[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]')

Afterwards we tried implementing restrictions on employee age, limiting employees to be at least 18 years old, but after further research, it was found that it was only possible through triggers, as it uses a now() function to get current time and CHECK constraints must always be deterministic according to the SQLite documentation.

5.1.10 Final SQLite database creation

The final SQLite database creation was, therefore, a mix of our original one with the ideas given by chatGPT.

After implementing all the constraints, it was possible to run the creation of the database, but the multiple DROP statements still wouldn't work if some tables (that had their primary keys pointing at other tables) were dropped before the "parent" table.

We addressed the problem by reordering our DROP statements and made it possible to run the query even when all tables were fully populated.

Initial order:

```
DROP TABLE IF EXISTS BusinessPartner;
DROP TABLE IF EXISTS Invoice;
DROP TABLE IF EXISTS Product;
DROP TABLE IF EXISTS ProductSale;
DROP TABLE IF EXISTS InvoiceSale;
```

```
DROP TABLE IF EXISTS Sale;
DROP TABLE IF EXISTS Contact;
DROP TABLE IF EXISTS Employee;
DROP TABLE IF EXISTS Person;
DROP TABLE IF EXISTS Role;
DROP TABLE IF EXISTS EmployeeDepartment;
DROP TABLE IF EXISTS Department;
DROP TABLE IF EXISTS Project;
DROP TABLE IF EXISTS Currency;
DROP TABLE IF EXISTS Purchase;
DROP TABLE IF EXISTS Company;
```

After reordering:

```
DROP TABLE IF EXISTS ProductSale;
DROP TABLE IF EXISTS InvoiceSale;
DROP TABLE IF EXISTS Product;
DROP TABLE IF EXISTS Sale;
DROP TABLE IF EXISTS EmployeeDepartment;
DROP TABLE IF EXISTS Project;
DROP TABLE IF EXISTS Invoice;
DROP TABLE IF EXISTS Contact;
DROP TABLE IF EXISTS Employee;
DROP TABLE IF EXISTS Purchase;
DROP TABLE IF EXISTS Department;
DROP TABLE IF EXISTS Currency;
DROP TABLE IF EXISTS Company;
DROP TABLE IF EXISTS Role;
DROP TABLE IF EXISTS BusinessPartner;
DROP TABLE IF EXISTS Person;
```

In the document “create2.sql” it is possible to see, commented, right before the current DROP statements, the original order we used before making this vital change.

5.1.11 Initial Data Loading

For the Initial Data Loading we decided to do the INSERT statements by hand, which proved a daunting task, once we had to ensure we were filling all the fields inside the constraints that were defined earlier.

After getting a first view of the implementation and understanding the need for a correct order to INSERT each table (by inserting the lesser dependent tables first), we developed the first populate file.

We decided to make a python script for the 2nd populate, which would insert meaningful and correct data, in large quantities in all tables, as creating 200 inserts would be a challenging and time consuming task.

The group opted for the use of the Faker library, to generate meaningful fake data on our tables for the “populate2.sql”, after doing the first populate, as this library allows for the insert of random data that matches the constraints encountered.

Example lines taken from “populate1.sql” file:

```
INSERT INTO BusinessPartner (Name, TaxID, Address) VALUES
('Not a Client Co.', '1234567890123456', '123 General Exemplo Street,
Porto'),
('Also Not a Fake Company ltd.', '9876543210987654', '122 A Street,
Lisbon');

INSERT INTO Company (Name, Address, Email, Phone) VALUES
('Super Company 1.', '456 Tech Park, Porto', 'info@sc1.com',
'+351123456789'),
('Lesser Super Company', '789 Logistics Ave, Lisbon', 'contact@lsc.com',
'+351987654321');

INSERT INTO Currency (CurrencyName) VALUES
('Euro'),
('US Dollar');
```

5.1.12 Generative AI Integration

ChatGPT 4.o was used to create the first crude script that was refined afterwards by the group.

The first prompt consisted in asking the AI to generate code examples for functions, using the Faker library, showcasing how to populate the database on demand.

It created helper functions to generate random emails and dates:

```
# Helper functions
def random_date(start, end):
    """Generate a random date between two dates."""
    return fake.date_between(start_date=start, end_date=end)

def random_email(name):
    """Generate a random email based on a name."""
    domain = fake.free_email_domain()
```

```
    return f"{name.lower().replace(' ', '')}@{domain}"
```

And examples on the actual implementation of the Faker library to populate a table:

```
def populate_business_partner():
    for _ in range(15):  # Add 15 business partners
        name = fake.name()
        tax_id = fake.ein()[:14]
        address = fake.address()
        cursor.execute("INSERT INTO BusinessPartner (Name, TaxID,
Address) VALUES (?, ?, ?)", (name, tax_id, address))
        write_to_file(f"INSERT INTO BusinessPartner (Name, TaxID,
Address) VALUES ('{name}', '{tax_id}', '{address}')")
```

5.1.13 Final Data Loading

We came across an error on tables that represented relations, where it tried to use ids that weren't real or that were already in use (when they should be unique).

Our solution was simple. We created a new function to get valid ids:

```
# Helper function to get a random ID from a table
def get_random_id(table_name, id_column):
    """Fetch a random ID from a given table and column."""
    cursor.execute(f"SELECT {id_column} FROM {table_name}")
    ids = cursor.fetchall()
    if ids:
        return random.choice(ids)[0]
    else:
        return None
```

It selects from an existing table and returns a random valid id.

For the population of Employee and Contact, we had to ensure there was a Person already created, and that an IDPerson would be used as the IDEmployee and IDContact. For that, we came with a simple solution. At first, we created a Contact table with a defined number of Contacts, afterwards we used those Contacts id's as a limit range to use as the IDEmployee and IDContact, ensuring that there won't be an Employee with: an ID already in use, an ID that is not in the Person table.

While the script populates the database, it also creates an output file, with all the INSERT statements needed to "repopulate" it, if needed (with the tables empty).

The final proposal can be found in the file "populate2.sql".

Qualitative Assessment of Group Member's Contribution

The division of tasks was done horizontally and all the members worked together on all the topics, whether in the research, drafting, writing or revision phase. All members contributed equally to the development of the project and the input of each one is reflected in all the pages written, be it in the actual report, in code or at the first drafting stages.

We tried taking advantage of each member's weaknesses and strengths, choosing, at each stage, tasks that could extract the greatest potential from each one. In the end, because of this division, all members helped on multiple facets of the project.